# NHSC/PACS Web Tutorials

Running the PACS Spectrometer pipeline for CHOP/NOD Mode

## PACS-301

*Level 0 to 1 processing*

*Line Scan and Short Range Scan Modes*

Updated Feb 2012

This tutorial assumes the following
version of HIPE

User Build 8.1 or later (CIB 8.0 3397 or later)

There are several improvements over previous
versions including calculating the responsivity from the on-orbit calibration
block taken before the observation
instead of using the pre-launch “nominal response”.
Additional more sophisticated flat fielding is done which
also includes refinements to the spectral “flat field”.

## Introduction

This tutorial will guide you through the interactive spectrometer pipeline from loading raw data into HIPE to obtain calibrated data with astrometry in the case of chop/nod mode.

## Pre-requisites

The following tutorials should be read before this one:

- ***PACS-101: How to use these tutorials.***
- ***PACS-102: Accessing and storing data from the*** *Herschel Science Archive*

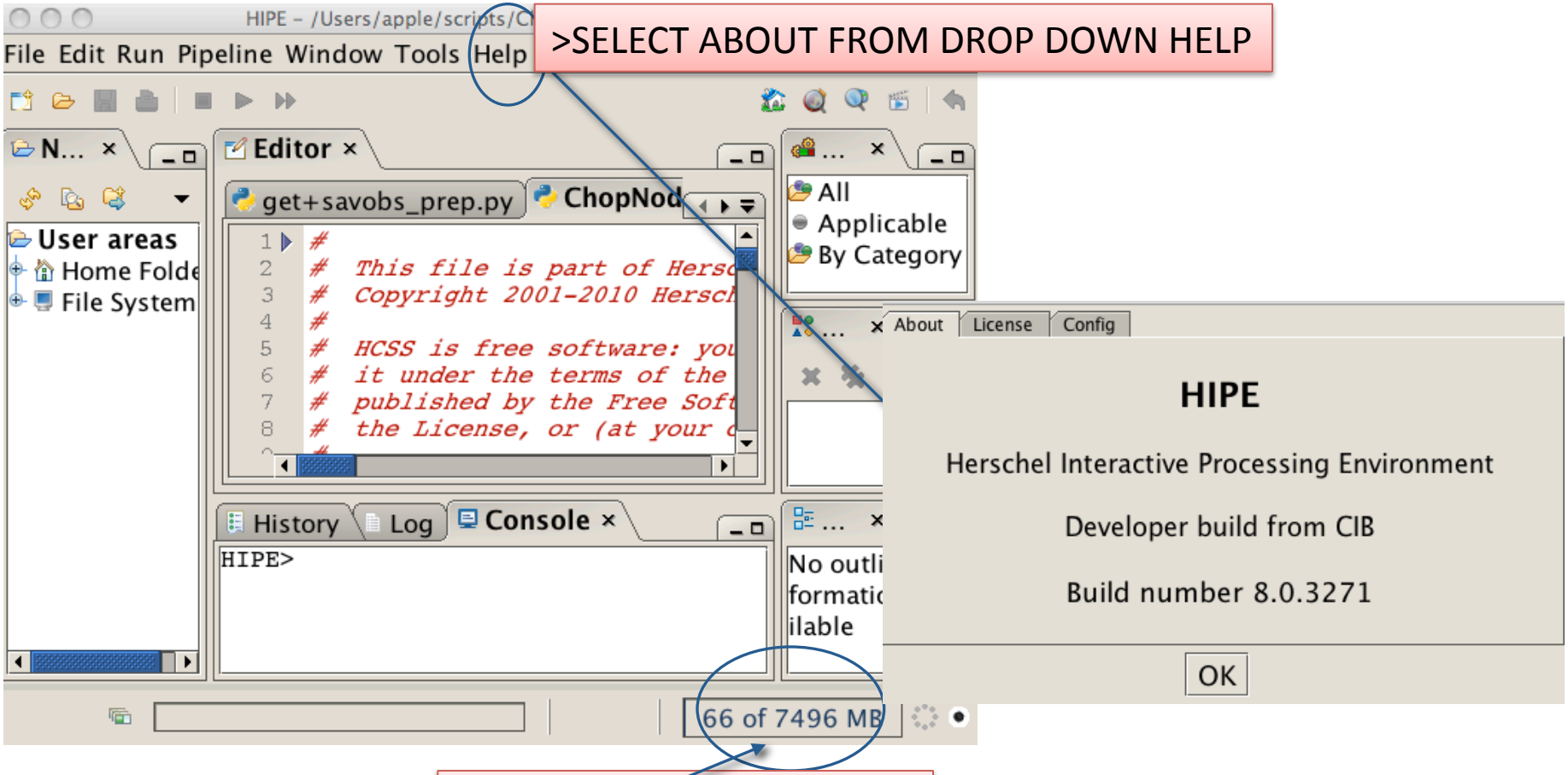
*Note that the following tutorial assumes that you have already created a pool in storage with the same name as the obsid of your observation. If this is not the case, consult PACS-102*

# Overview

**Step 1** Check HIPE version and memory
**Step 2** Setup (Script and load data)
**Step 3** Read in data from script (optional)
**Step 4** Check Observation Context and set verbose levels etc.
**Step 5** Run the 0 → 0.5 pipeline
**Step 6** Run the 0.5 → 1 pipeline

**Step 1:** Check that you are running the appropriate version of HIPE for this Tutorial and check the memory

HOW TO DETERMINE YOUR BUILD NUMBER AND HOW MUCH MEMORY IS ALLOCATED. Always allocate less than the RAM capacity of your machine—this example is an 8GByte Laptop so I allocated 7.5GBtyes

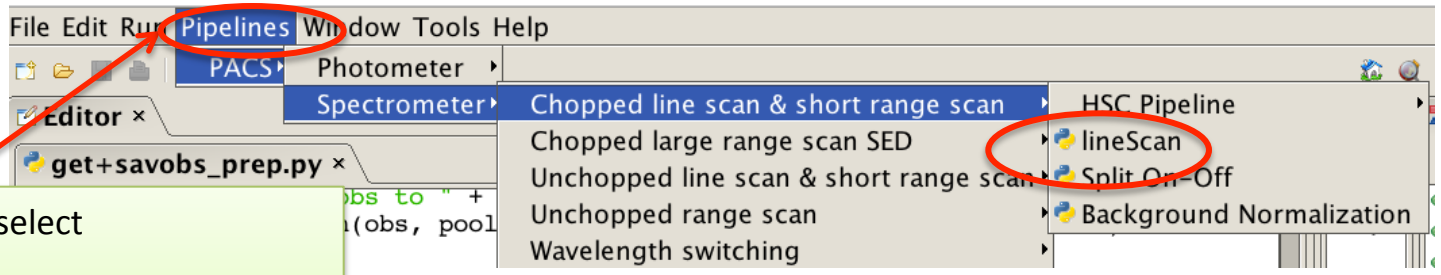



(Allocate in File>Preferences>general>Start up and Shut Down)+restart HIPE

# Step 2:

Load in the correct script for processing your data

In this example we assume you are not using your own specialist script. Here we take the script from the “pipeline” pulldown menu




From the Pipeline Menu select

PACS/Spectrometer/Chopped line scan & short range scan/lineScan

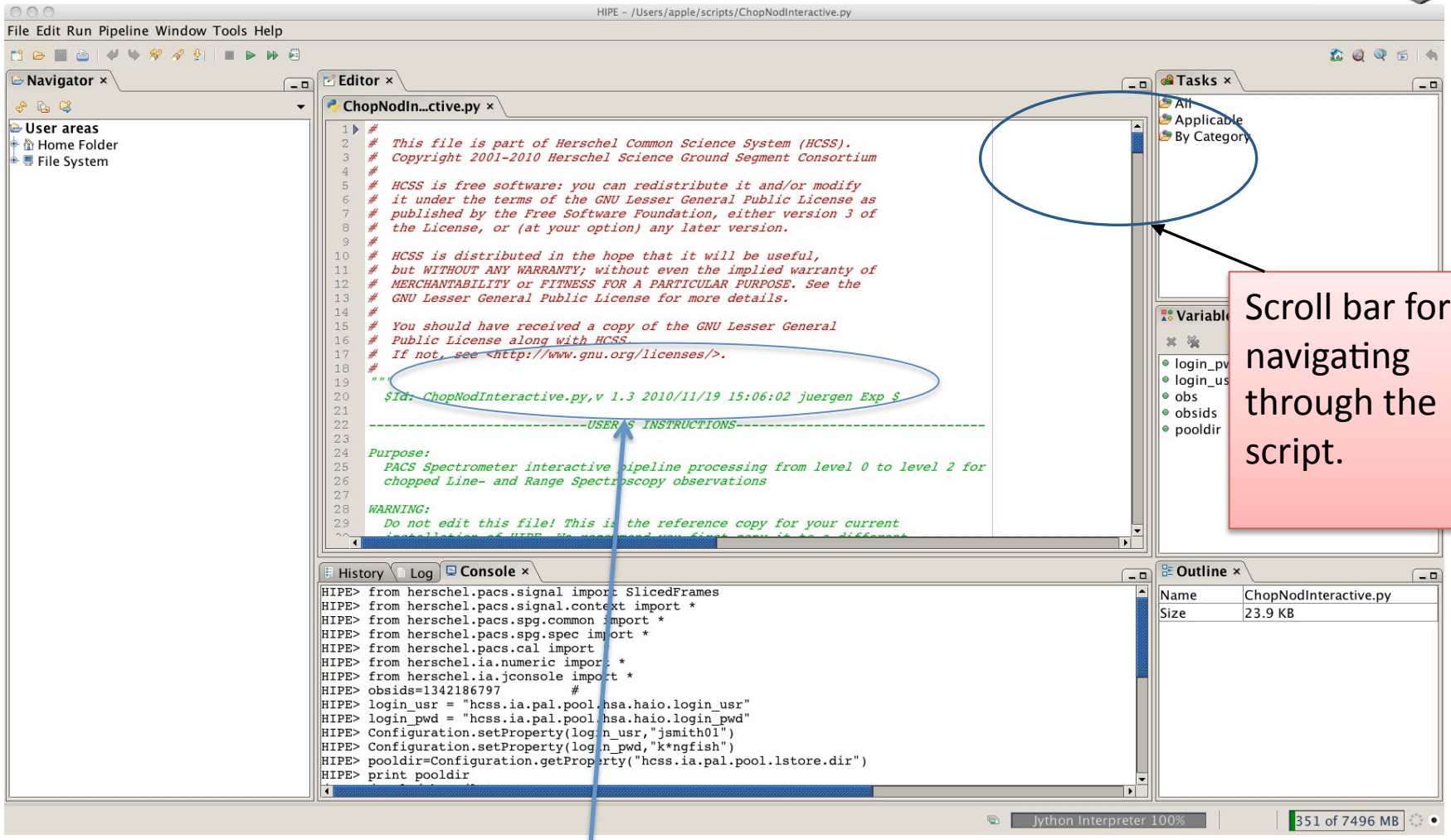
Note that there are several options in the Chop/Nod mode.

HSC Pipeline are the scripts used by the HSC pipeline and are intended to be non-interactive

Split ON-OFF allows the user to make separate ON and OFF maps and SHOULD NOT BE USED FOR DATA ANALYSIS
Its purpose is to allow the user to inspect the “chop off” positions for contamination by a source if suspected.

Background Normalization is an alternative way to calibrate your data using the telescope background from the chop-off observation to calibrate these data. The standard scrip uses the calibration block. This alternative method is only recommended when the observer is interested in measuring accurately a faint continuum level—for example if you have a faint absorption line and you wish to measure accurately the equivalent width of the line. In most other respects the standard pipeline works well.

This should look like something like this when read in:


HIPE - /Users/apple/scripts/ChopNodInteractive.py
File Edit Run Pipeline Window Tools Help
Navigator
User areas
Home Folder
File System
Editor
ChopNodIn...ctive.py
# This file is part of Herschel Common Science System (HCSS).
# Copyright 2001-2010 Herschel Science Ground Segment Consortium
# HCSS is free software: you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License as
# published by the Free Software Foundation, either version 3 of
# the License, or (at your option) any later version.
# HCSS is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU Lesser General Public License for more details.
# You should have received a copy of the GNU Lesser General
# Public License along with HCSS.
# If not, see <http://www.gnu.org/licenses/>.
$Id: ChopNodInteractive.py,v 1.3 2010/11/19 15:06:02 juergen Exp $
----------USER'S INSTRUCTIONS----------
Purpose:
PACS Spectrometer interactive pipeline processing from level 0 to level 2 for
chopped Line- and Range Spectroscopy observations
WARNING:
Do not edit this file! This is the reference copy for your current
Tasks
All
Applicable
By Category
Variabl
login_pw
login_us
obs
obsids
pooldir
History
Log
Console
HIPE> from herschel.pacs.signal import SlicedFrames
HIPE> from herschel.pacs.signal.context import *
HIPE> from herschel.pacs.spg.common import *
HIPE> from herschel.pacs.spg.spec import *
HIPE> from herschel.pacs.cal import *
HIPE> from herschel.ia.numeric import *
HIPE> from herschel.ia.jconsole import *
HIPE> obsids=1342186797 #
HIPE> login_usr = "hcss.ia.pal.pool.hsa.haio.login_usr"
HIPE> login_pwd = "hcss.ia.pal.pool.hsa.haio.login_pwd"
HIPE> Configuration.setProperty(login_usr,"jsmith01")
HIPE> Configuration.setProperty(login_pwd,"k*ngfish")
HIPE> pooldir=Configuration.getProperty("hcss.ia.pal.pool.lstore.dir")
HIPE> print pooldir
Outline
Name ChopNodInteractive.py
Size 23.9 KB
Jython Interpreter 100%
351 of 7496 MB

Scroll bar for navigating through the script.

The file you have selected should be named in the script

# Step 3

Execute the first few lines of the script to load up your observational context “obs” from a pool. Unlike previous versions, the script now includes steps which will load “obs” from a pre-existing pool.

NOTE: If you prefer, you can skip these first few lines and read “obs” into your session using PACS tutorial PACS-202. If you have already done this then skip to STEP 4. Otherwise continue.

Insert the correct obsid. Then just run the following lines of the script if a) you already have a pool in the default location ~/.hcss/lstore and b) if the pool has the same name as the obsid (recommended)

```
# will be present if you are running the multiObs script. If multi
# present, the obsid will have been set already, and if not then t
# here. (If you get a NameError, then the obsid had not been set.)
if ((not locals().has_key('multiObs')) or (not multiObs)):
        obsid = 1342188943

useHsa = 0
obs    = getObservation(obsid, verbose=True, useHsa=useHsa, poolLocation=None, poolName=None)
if useHsa: saveObservation(obs, poolLocation=None, poolName=None)
```

Otherwise, either set the poolLocation, and poolName variables to the location of your pool before running this part

poolLocation='~/mypooldir/', poolname='mynamepool'

If this is the first time you have accessed your data (i.e. you have not already created a data pool) then please follow the instructions in Pacs tutorial PACS-102.

**Step 4**: Check that you have indeed loaded the correct Observational Context

# Information Box:

**What is an Observation Context?**

To use an interactive (ipipe) scripts, the observation context must be defined in a variable named **obs**

- *The observation context provides the link to the data and all relevant ancillary and calibration products. It is a container holding all this material in one place*

***Documentation References**:*

- *HIPE user manual Version 1.2, chapter 3, pgs. 45-56*
- *PACS data reduction guide Version 3, Section 5.2, pg. 145*

## Check # 1

The observation context is properly loaded in HIPE.

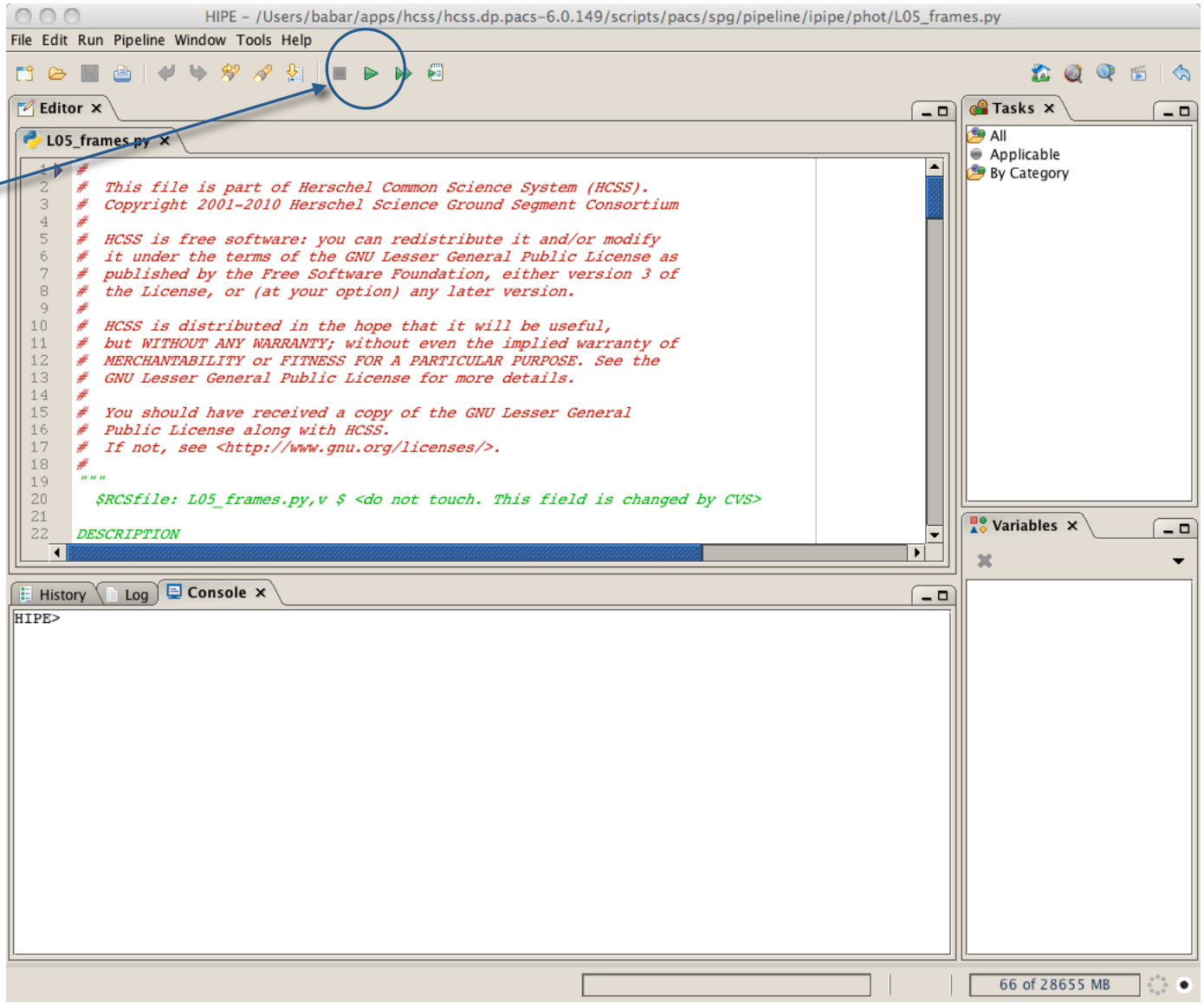
**Type in console window**

Observation ID and data type are identified.

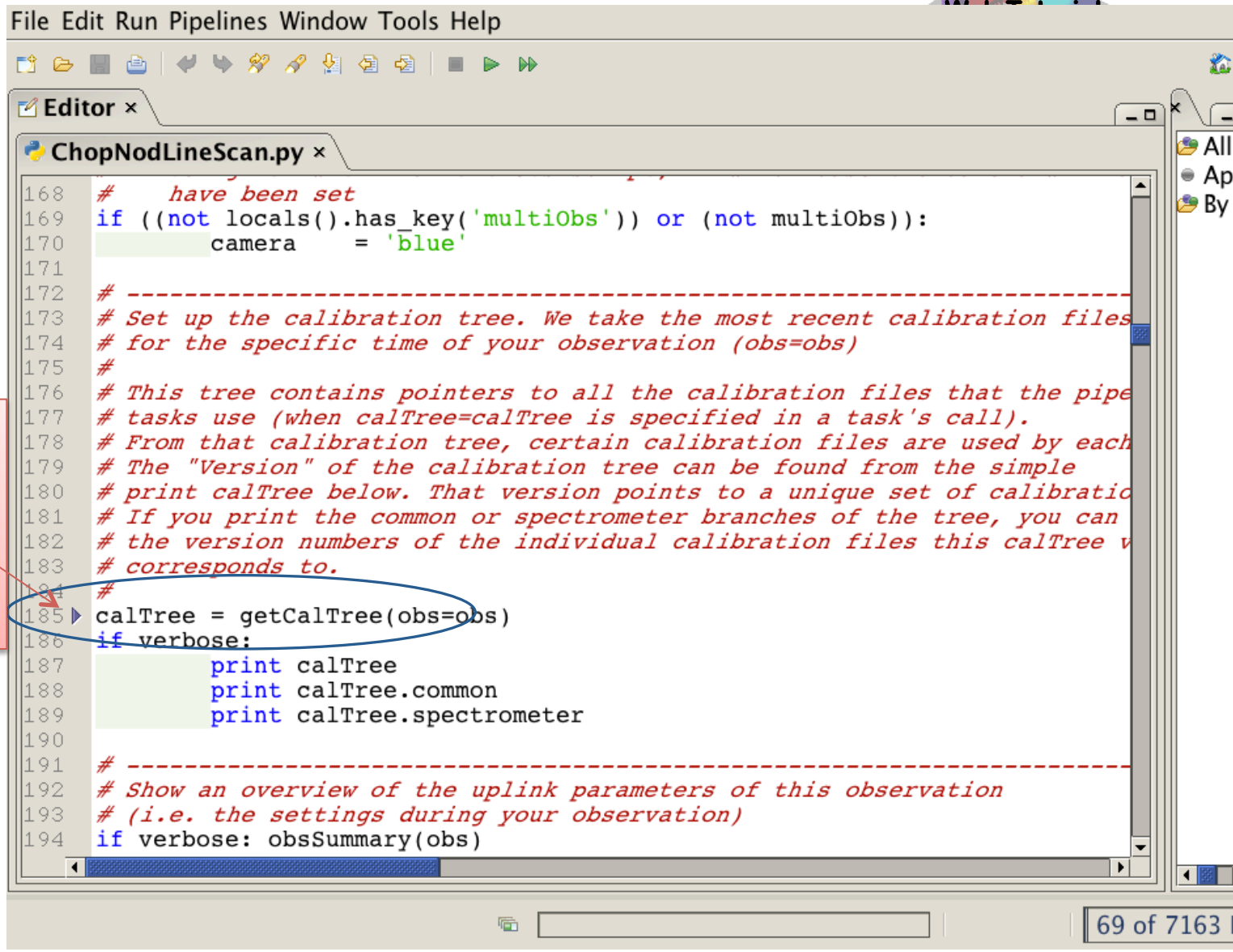
**Output looks similar to one shown here.**

```
HIPE> print obs
{description="ObservationContext for PACS data of observation 1342186797", meta=[type, creator, creationDate, description, instrument, modelName,
startDate, endDate, formatVersion, obsState, obsid, odNumber, cusMode, instMode, obsMode, processingMode, origin, slewTime, aorLabel, aot, equinox,
missionConfig, object, raDeSys, pmRA, pmDEC, raNominal, decNominal, ra, dec, posAngle, telescope, velocityDefinition, radialVelocity, observer,
proposal, pointingMode, missionConfiguration, instrumentConfiguration, startTime, endTime, chopAvoidFrom, chopAvoidTo, chopNod, decoff, doSlewScience,
faintLines, fluxUnit, lWave, lcontFlux, lineFlux, lineId, lineStep, lineWidth, lines, m, mapRasterAngle, n, naifid, obsOverhead, orderSel, pointStep,
raoff, redshiftType, redshiftValue, refSelected, repeatLine, source, throw, userNODcycles, widthUnit, subType, compVersion, algoNumber, algorithm,
compNumber, compMode, dxid], datasets=[], history=None,
refs=[auxiliary,calibration,level0,level0_5,level1,level2,logObsContext,quality,qualitySummary]}
HIPE>
```

Click on the single green arrow to step through the commands in the script (Recommended for first time users)



As you click the green arrow the small black arrow will move down through the script


File Edit Run Pipelines Window Tools Help
Editor
ChopNodLineScan.py
168 # have been set
169 if ((not locals().has_key('multiObs')) or (not multiObs)):
170 camera = 'blue'
171
172 # ----------
173 # Set up the calibration tree. We take the most recent calibration files
174 # for the specific time of your observation (obs=obs)
175 #
176 # This tree contains pointers to all the calibration files that the pipe
177 # tasks use (when calTree=calTree is specified in a task's call).
178 # From that calibration tree, certain calibration files are used by each
179 # The "Version" of the calibration tree can be found from the simple
180 # print calTree below. That version points to a unique set of calibratio
181 # If you print the common or spectrometer branches of the tree, you can
182 # the version numbers of the individual calibration files this calTree v
183 # corresponds to.
184 #
185 calTree = getCalTree(obs=obs)
186 if verbose:
187 print calTree
188 print calTree.common
189 print calTree.spectrometer
190
191 # ----------
192 # Show an overview of the uplink parameters of this observation
193 # (i.e. the settings during your observation)
194 if verbose: obsSummary(obs)
All
Ap
By
69 of 7163

SET THE VERBOSE LEVEL AND SPECTROMETER "Camera". This will govern whether you get a red or blue spectrum analyzed so its important because some calibration needs to know which side is being analysed. You will need to repeat the whole exercise for the other channel if you want to analyse both channels.

VERBOSE = 1 turns on diagnostic plots and other useful info

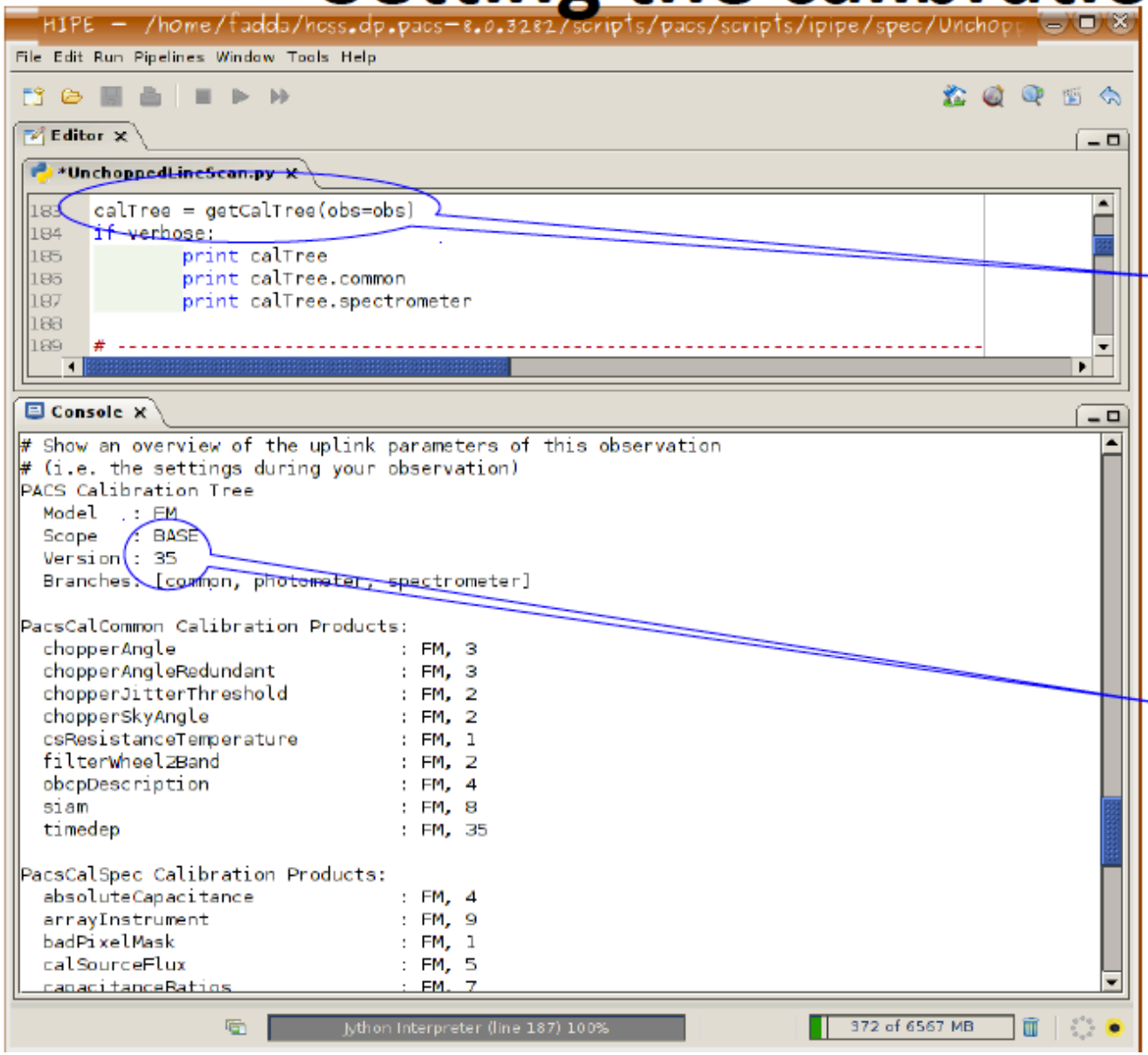
```
verbose = 1

# ------------------------------------------------------------------
# SETUP 0.5
# updateObservationContext
#  At the end of each level, do you want to update the Observation
#  choice:
#          0 - do not update the observation context
#          1 - update the observation context
#  See further down the script for simple ways to save your reduct
#  at various steps in the data processing
updateObservationContext = 0

# ------------------------------------------------------------------
# SETUP 1:
#    - Red or blue camera ? As before, we test for whether this sc
#    being run within a multiObs script, in which case the camera
#    have been set
if ((not locals().has_key('multi
        camera     = 'blue'
"
```

set the band you want 'red' of 'blue' depending on the line
You can always execute
>print camera to check later

# Setting the calibration tree




Finally, we set the calibration tree.

Read the time stamp of our obs and apply the calibration from the used distribution.

Version 32 and later ones consider the calibration block and include improved flat field corrections.

Now print a summary of your data. This command will print out a lot of useful information about your observation. Note that since you have already set the camera to either red or blue, then only the scans observed in that band are shown. If you made a mistake and meant to analyze data from the other band, then now is the time to go back and start over with the correct camera selected. The command assumes verbose = 1

```
if verbose: obsSummary(obs)
```

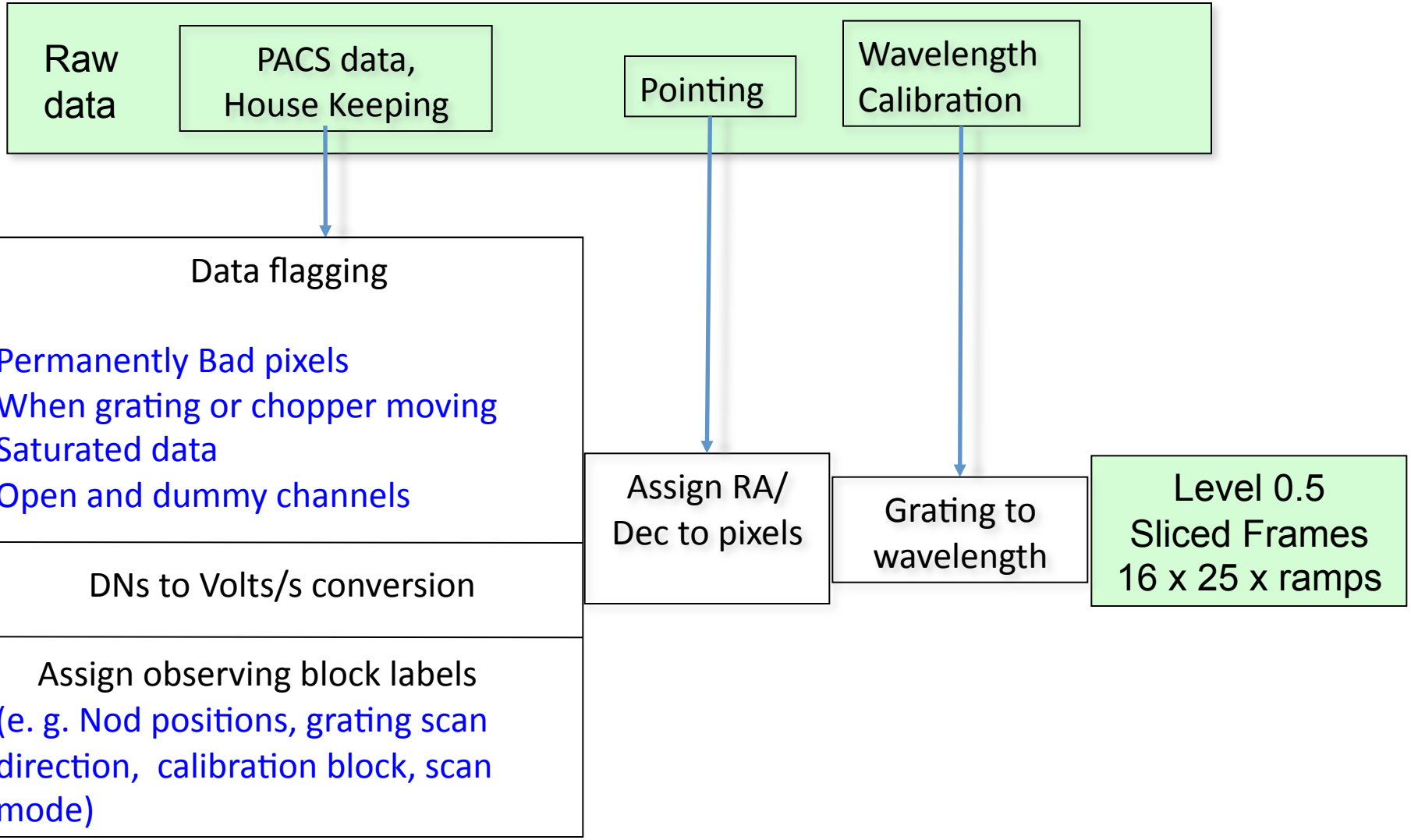
Note that you can just type obsSummary (obs) in the console window at any time in your session to get the summary

# Step 5

## Run the 0 → 0.5 pipeline

Basic calibration (pointing, wavelength calibration, slicing)

# Level 0 → 0.5


Raw data
PACS data, House Keeping
Pointing
Wavelength Calibration
Data flagging
Permanently Bad pixels
When grating or chopper moving
Saturated data
Open and dummy channels
DNs to Volts/s conversion
Assign observing block labels
(e. g. Nod positions, grating scan direction, calibration block, scan mode)
Assign RA/ Dec to pixels
Grating to wavelength
Level 0.5
Sliced Frames
16 x 25 x ramps

# Extracting level-0 data

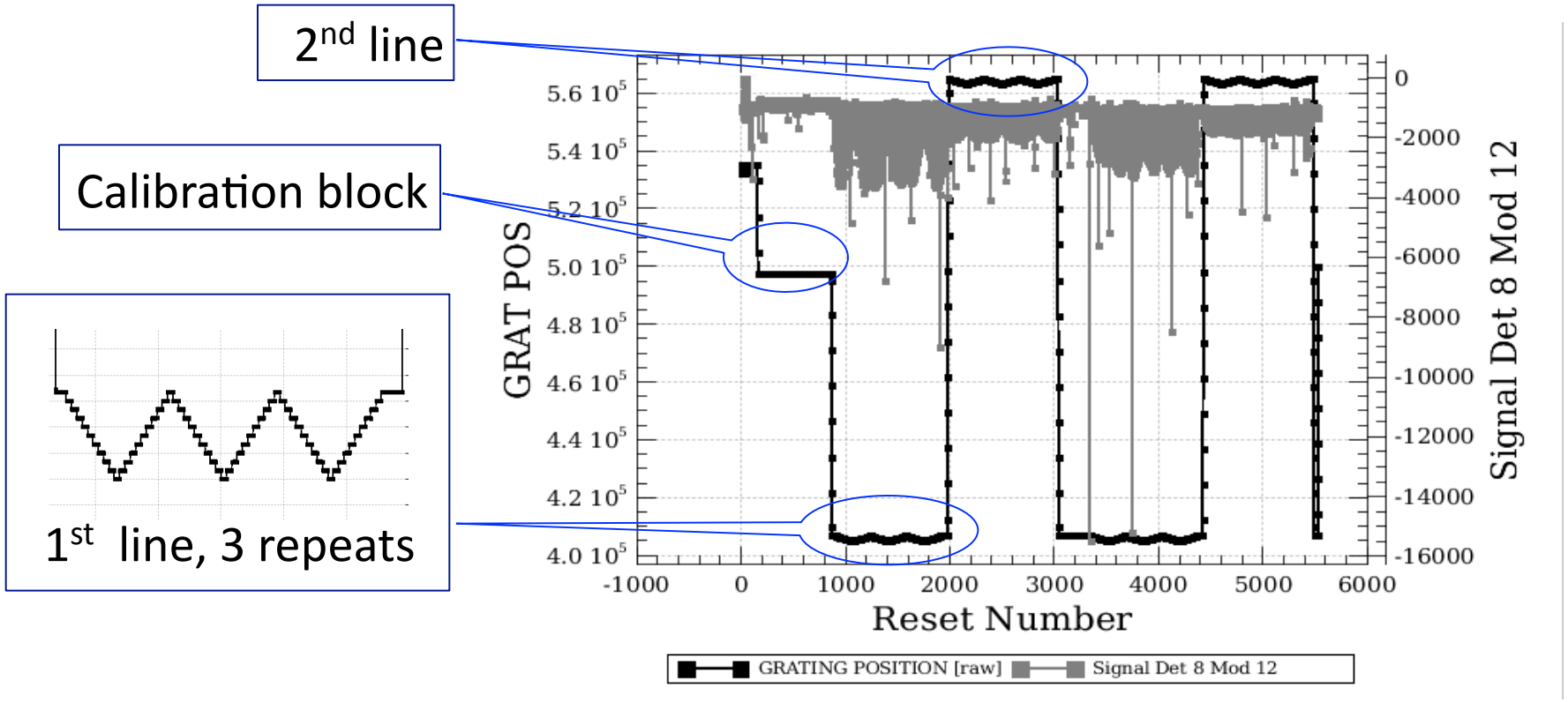
ChopNodEx...RSION.py

```
164
165 # ------------------------------------------------------------------------
166 # Extract the level-0 products from the ObservationContext
167 #       Get and prepare the level0 product that you will pipeline on.
168 #       First we copy the metadata from the ObservationContext to the level0 product
169 #       Then we extract the level0 product from the ObservationContext
170 pacsPropagateMetaKeywords(obs,'0', obs.level0)
171 level0 = PacsContext(obs.level0)
172
173 # For your camera, extract the Frames (scientific data), the rawramps (raw data
174 # for one pixel), and the DMC header (the mechanisms' status information,
175 # sampled at a high frequency)
176 slicedFrames  = SlicedFrames(level0.fitted.getCamera(camera).product)
177 slicedRawRamp = level0.raw.getCamera(camera).product
178 slicedDmcHead = level0.dmc.getCamera(camera).product
179
180 if verbose:
181         # Get an overview of the basic structure of the data, prior to any processing
182         slicedSummary(slicedFrames)
183         # Plot the grating position & raw signal of central pixel
184         p0 = slicedSummaryPlot(slicedFrames,signal=1)
185
```

From now on, we will step through the script line by line using the green arrow on the menu bar. The first step consists in extracting the 0-level products from the observation context.

# Check: level 0

Let's check first the basic structure of the data. It is important to verify if the observation has been performed correctly.




In our case, after the calibration block, we can identify two different lines observed 3 times in the two nod positions.

# Some flagging

ChopNodEx...RSION.py

```
186  # ------------------------------------------------------------------------
187  #        Processing      Level 0 -> Level 0.5
188  # ------------------------------------------------------------------------
189
190
191  # flag the saturated data in a mask "SATURATION" (and "RAWSATURATION": this
192  # uses the raw data we get for some pixels)
193  # used cal files: RampSatLimits and SignalSatLimits
194  slicedFrames = specFlagSaturationFrames(slicedFrames, rawRamp = slicedRawRamp, calTree=calTree)
195
196  # Convert digital units to Volts, used cal file: Readouts2Volts
197  slicedFrames = specConvDigit2VoltsPerSecFrames(slicedFrames, calTree=calTree)
198
199  # Identify the calibration blocks and fill the CALSOURCE Status entry
200  slicedFrames = detectCalibrationBlock(slicedFrames)
201
202  # Add the time information in UTC to the Status
203  slicedFrames = addUtc(slicedFrames, obs.auxiliary.timeCorrelation)
204
```

# Adding pointing products

ChopNodEx...RSION.py

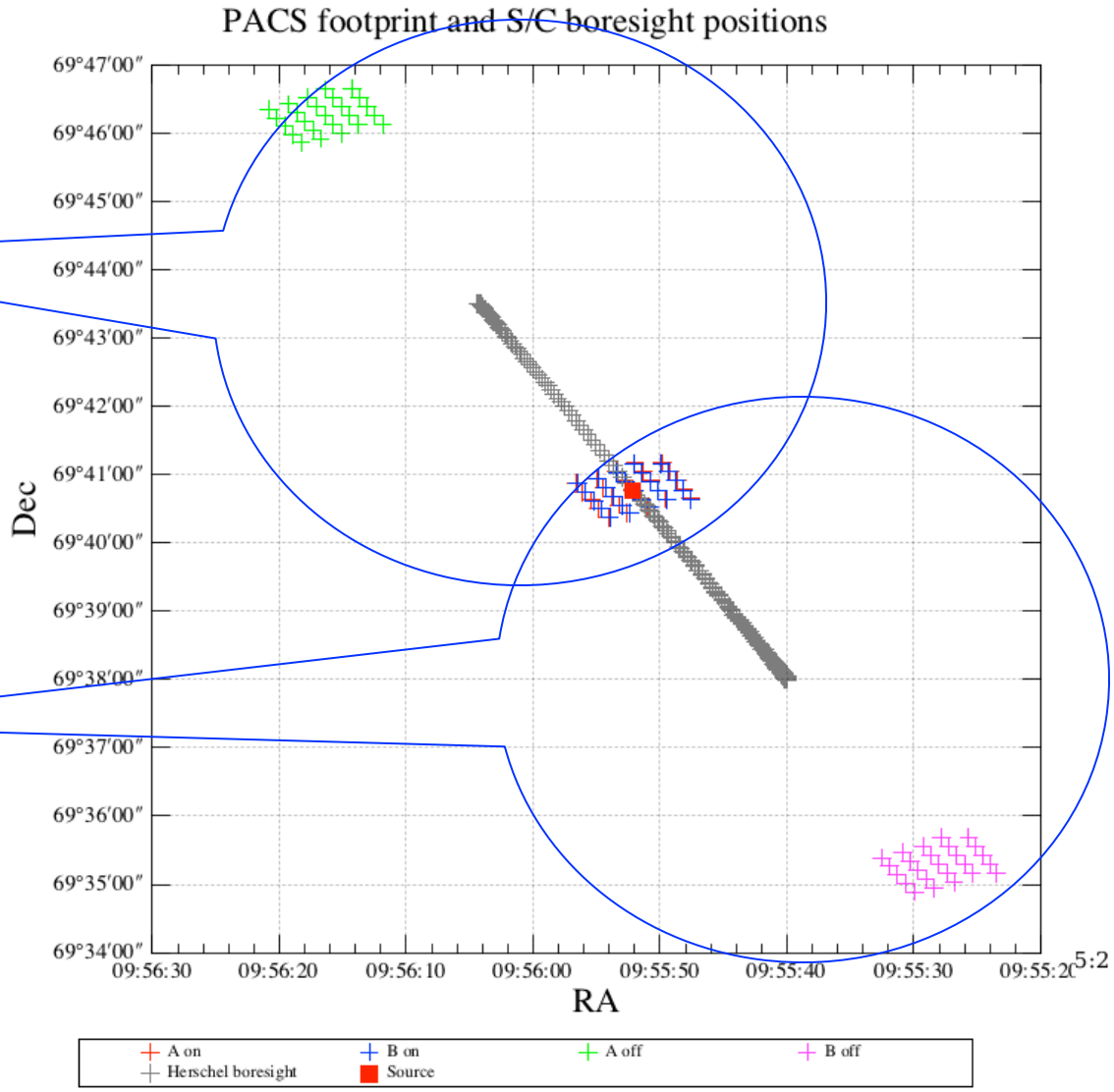
```
204
205  # Add the pointing information to the Status
206  #   Uses the pointing, horizons product (solar system object ephemeries),
207  #   orbitEphemeris products, and the SIAM cal file.
208  # NOTE for SSOs: you should contact the HSC helpdesk to check whether the
209  #   pointing is currently correctly handled for moving objects.
210  slicedFrames = specAddInstantPointing(slicedFrames, obs.auxiliary.pointing, calTree = calTree, orbit
211
212  # Extend the Status of Frames with the parameters GRATSCAN, CHOPPER, CHOPPOS
213  # used cal file: ChopperThrowDescription
214  slicedFrames = specExtendStatus(slicedFrames, calTree=calTree)
215
216  # Convert the chopper readouts to an angle wrt. focal plane unit and the sky
217  # and add this to the Status, used cal files: ChopperAngle and ChopperSkyAngle
218  slicedFrames = convertChopper2Angle(slicedFrames, calTree=calTree)
219
220  # Add the positions for each pixel (Ra and Dec datasets)
221  # used cal files: ArrayInstrument and ModuleArray
222  slicedFrames = specAssignRaDec(slicedFrames, calTree=calTree)
223
224  if verbose:
225          # show footprint of your observation
226          slicedPlotPointing(slicedFrames)
```

At this point, astrometry is added to the data. At each pixel a coordinate is assigned and we can check if the pointing was performed correctly.

# Check: footprint

Nod A

Nod B



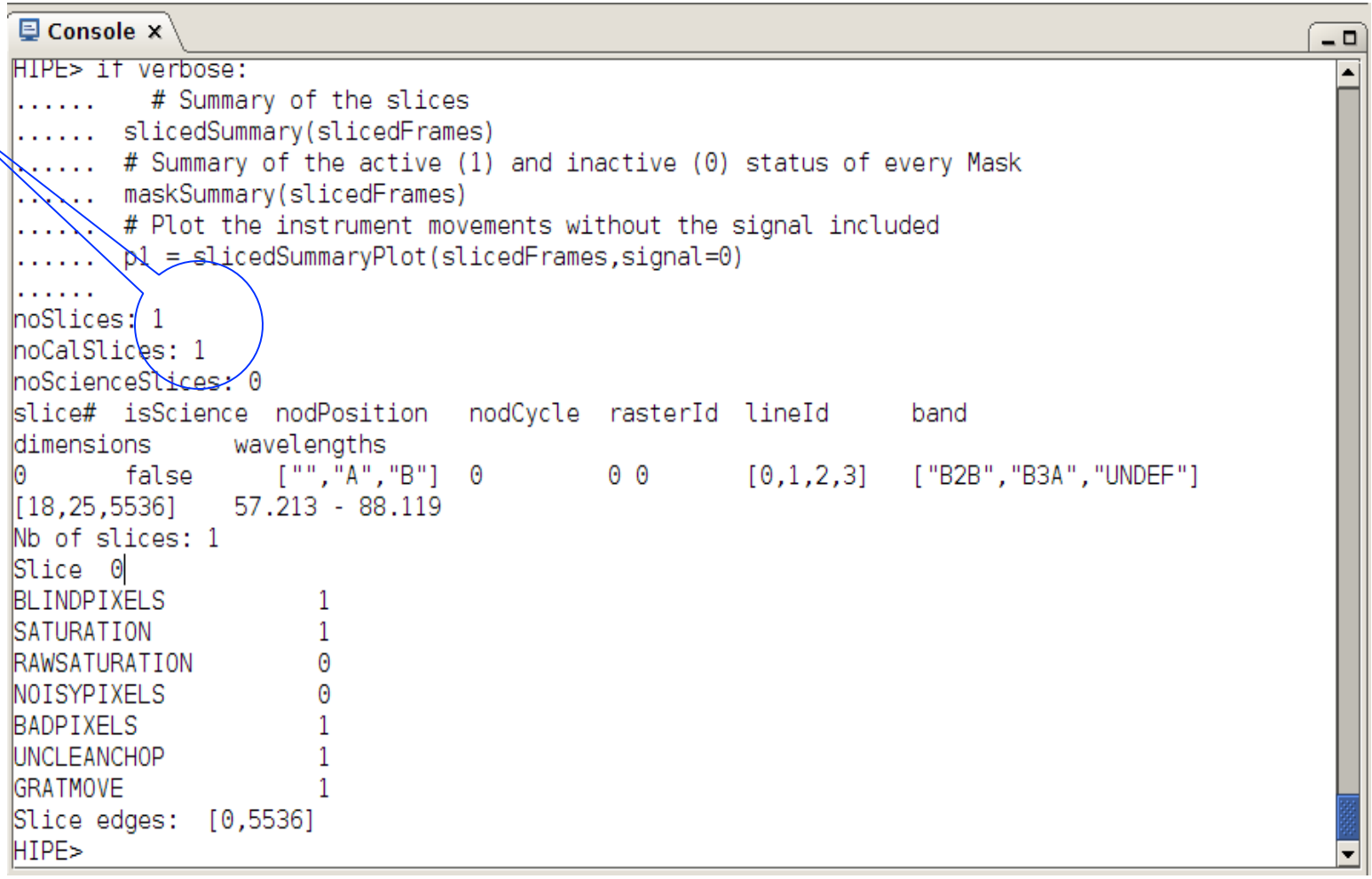
# Masks and wav. calibration

ChopNodEx...RSION.py

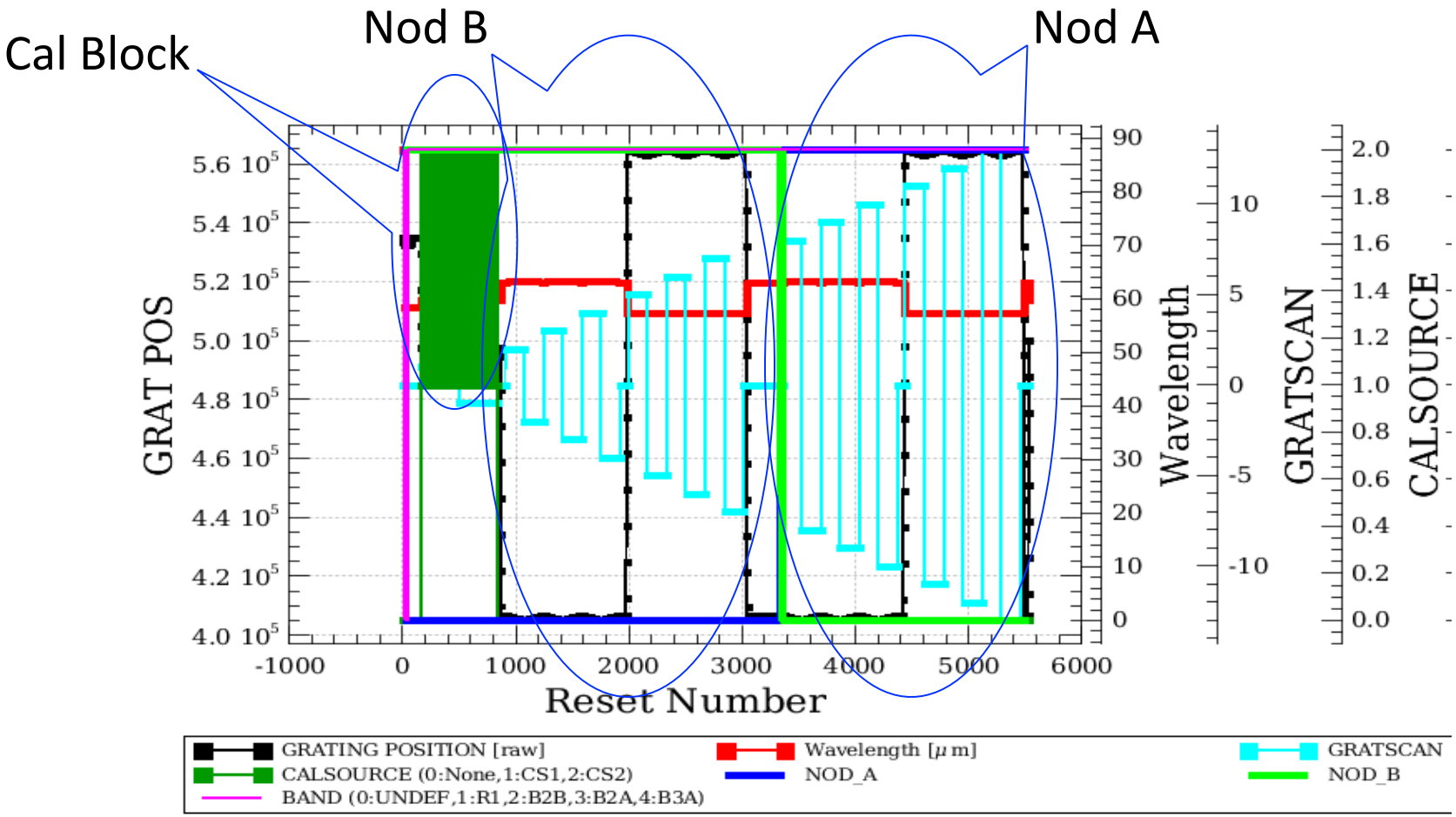
```
228 # Add the wavelength for each pixel (Wave dataset), used cal file: WavePolynomes
229 slicedFrames = waveCalc(slicedFrames, calTree=calTree)
230
231 # Correct the wavelength for the spacecraft velocity.
232 # Uses the pointing, orbitEphemeris and timeCorrelation product.
233 slicedFrames = specCorrectHerschelVelocity(slicedFrames, obs.auxiliary.orbitEphemeris, obs.auxiliary
234
235 # Find the major logical blocks of this observation and organise them in the
236 # BlockTable attached to the Frames; used cal file: ObcpDescription
237 slicedFrames = findBlocks(slicedFrames, calTree = calTree)
238
239 # Flag the known bad or noisy pixels in the masks "BADPIXELS" and "NOISYPIXELS"
240 # used cal files: BadPixelMask and NoisyPixelMask
241 #   -> by default the bad pixels will be excluded later when final cubes are built, the noisy pixels
242 slicedFrames = specFlagBadPixelsFrames(slicedFrames, calTree=calTree)
243
244 # Flag the data affected by the chopper movement in the mask "UNCLEANCHOP"
245 # Uses the high resolution Dec/Mec header and the cal files ChopperAngle and ChopperJitterThreshold
246 slicedFrames = flagChopMoveFrames(slicedFrames, dmcHead=slicedDmcHead, calTree=calTree)
247
248 # Flag the data effected by the grating movement in the mask "GRATMOVE"
249 # Uses the high resolution Dec/Mec header and the cal file GratingJitterThreshold
250 slicedFrames = flagGratMoveFrames(slicedFrames, dmcHead=slicedDmcHead, calTree=calTree)
251
252 if verbose:
253     # Summary of the slices
254       hgyr6745454te     slicedSummary(slicedFrames)
255         # Summary of the active (1) and inactive (0) status of every Mask
256         maskSummary(slicedFrames)
257         # Plot the instrument movements without the signal included
258         p1 = slicedSummaryPlot(slicedFrames,signal=0)
```

# Check: before slicing

Only 1 slice

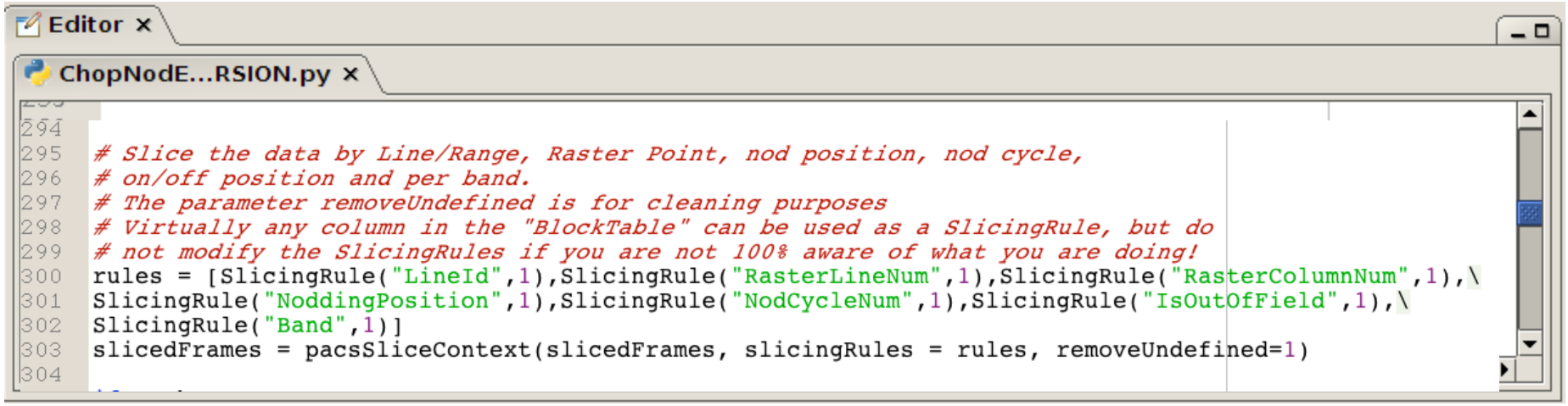


# Check: before slicing




There are two lines (two wavelengths in red). Grating scans are numbered positive if upscans and negative if downscans.

# Slicing


Editor
ChopNodE...RSION.py
294
295 # Slice the data by Line/Range, Raster Point, nod position, nod cycle,
296 # on/off position and per band.
297 # The parameter removeUndefined is for cleaning purposes
298 # Virtually any column in the "BlockTable" can be used as a SlicingRule, but do
299 # not modify the SlicingRules if you are not 100% aware of what you are doing!
300 rules = [SlicingRule("LineId",1),SlicingRule("RasterLineNum",1),SlicingRule("RasterColumnNum",1),\
301 SlicingRule("NoddingPosition",1),SlicingRule("NodCycleNum",1),SlicingRule("IsOutOfField",1),\
302 SlicingRule("Band",1)]
303 slicedFrames = pacsSliceContext(slicedFrames, slicingRules = rules, removeUndefined=1)
304

The slicing of the data is performed according to rules made explicit in the pipeline. In our example, two lines are observed in two nodding positions. So, we expect 4 slices plus an initial slice containing the calibration block.

# Check: after slicing

5 slices !

Line 1 – B & A nodes

Line 2 – B & A nodes

Console ×

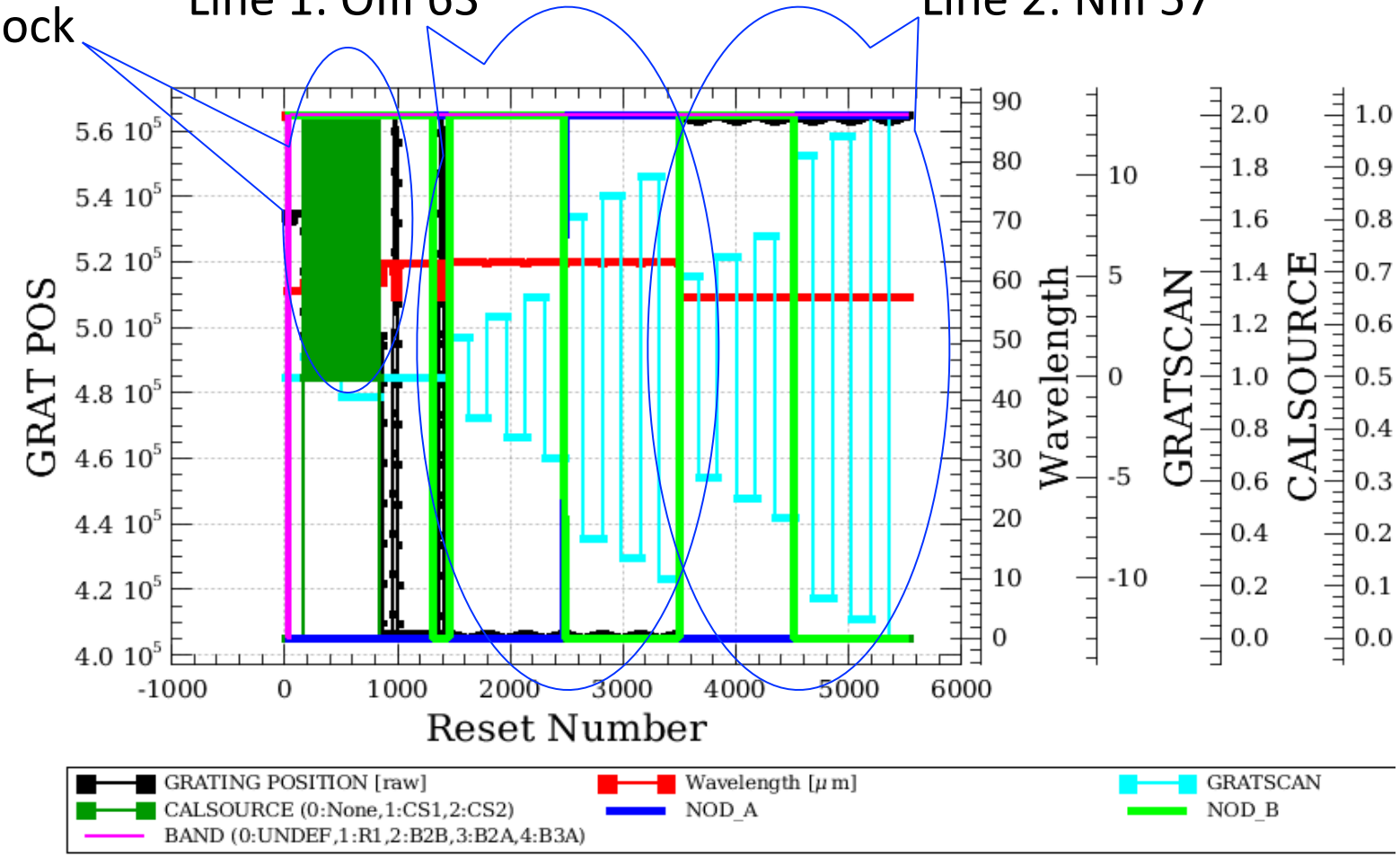
```
HIPE> rules =
[SlicingRule("LineId",1),SlicingRule("RasterLineNum",1),SlicingRule("RasterColumnNum",1),Slici
ngRule("NoddingPosition",1),SlicingRule("NodCycleNum",1),SlicingRule("IsOutOfField",1),Slicing
Rule("Band",1)]
HIPE> slicedFrames = pacsSliceContext(slicedFrames, slicingRules = rules)
HIPE> if verbose:
.....  slicedSummary(slicedFrames)
.....  p2 = slicedSummaryPlot(slicedFrames,signal=0)
.....
noSlices: 5
noCalSlices: 1
noScienceSlices: 4
slice#  isScience  nodPosition   nodCycle  rasterId  lineId      band
dimensions      wavelengths
0       false      ["","B"]      0         0 0       [0,1]       ["B2B","B3A","UNDEF"]
[18,25,1460]    57.213 - 88.119
1       true       ["B"]         1         0 0       [2]         ["B3A"]
[18,25,1019]    63.093 - 63.379
2       true       ["A"]         1         0 0       [2]         ["B3A"]
[18,25,1019]    63.093 - 63.379
3       true       ["B"]         1         0 0       [3]         ["B3A"]
[18,25,1019]    57.213 - 57.548
4       true       ["A"]         1         0 0       [3]         ["B3A"]
[18,25,1019]    57.213 - 57.548
Slice edges:  [0,1460,2479,3498,4517,5536]
HIPE>
```

# Check: after slicing

Cal Block

Line 1: OIII 63

Line 2: NIII 57




There are four slices (calibration, nod A and B for the 1st line, nod A and B for the 2nd line).

# Step 5

## Run the 0.5 → 1 pipeline

Glitch detection, chop differentiation, RSRF, flat

# Level 0.5 → 1

Level 0.5

Glitch detection

Use Cal Block to measure responsivity and dark

Subtract On and Off Chop Signals

Divide by Relative Spectral Response Function (RSRF)

Apply Responsivity Correction derived from Cal Block

Convert Data Frame to Cube

Apply Flat Field

Level 1
Frames to Cubes
(5 x 5 x (16 x ramps) )

# Deglitching

ChopNodEx...RSION.py

```
298  # De-activate all masks before running the glitch flagging
299  slicedFrames = activateMasks(slicedFrames, String1d([" "]), exclusive = True)
300
301  if verbose: maskSummary(slicedFrames,slice=0)
302
303  # Detect and flag glitches ("GLITCH" mask)
304  slicedFrames = specFlagGlitchFramesQTest(slicedFrames)
305
306
307  if verbose:
308          slicedSummary(slicedFrames)
309      # Summary plot, including the signal
310          p3 = slicedSummaryPlot(slicedFrames,signal=1)
311          # Plot of signal vs wavelength for the central pixel for a single slice (you can chose any s
312          # Detector signal: you will see the on and off chop spectral data together
313          # on this plot, as they have not been subtracted from each other yet
314          # (compare this plot to p6 below)
315          slice = 1
316          p4 = plotSignalBasic(slicedFrames, slice=slice)
317          # Inspect timeline of signals and masked signals via a viewer
318          MaskViewer(slicedFrames.get(slice))
```

At this point, a deglitching code creates a glitch mask. Sometimes, this deglitching can be too aggressive. It is therefore possible to ignore this mask and detect outliers using data redundancy (see tutorial level 1 → level 2)

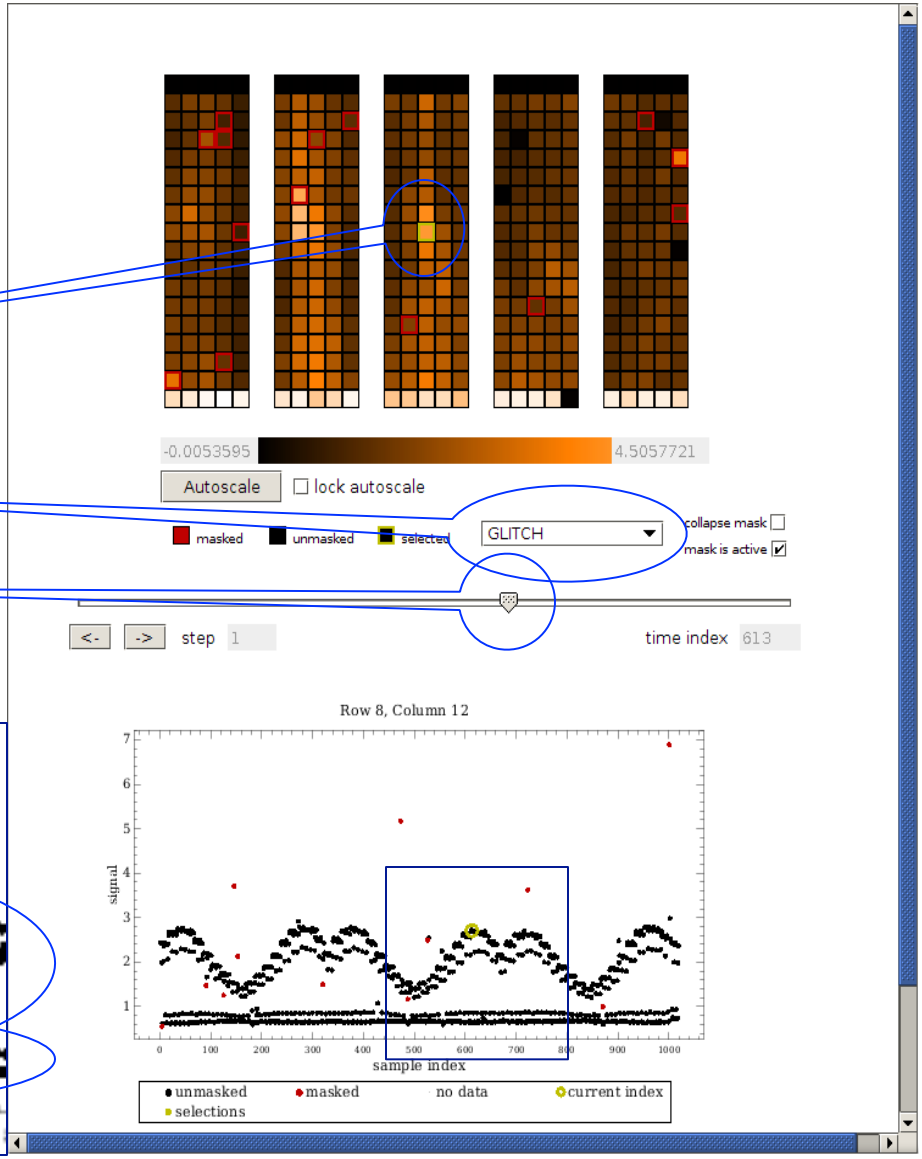
# Glitch detection

You can check manually the points flagged as glitch or masked for other reasons using the maskviewer

Select a pixel by clicking on it

Select a mask

Select a frame

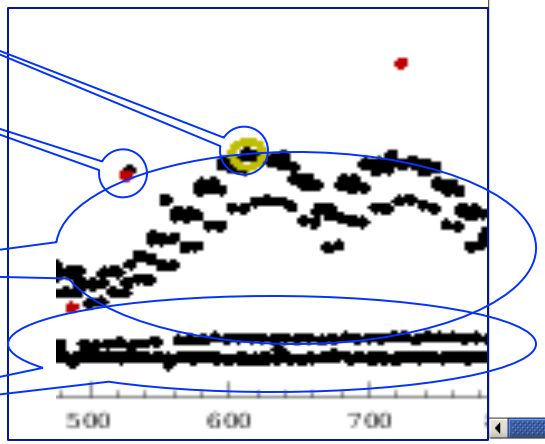



Current frame

Masked glitch

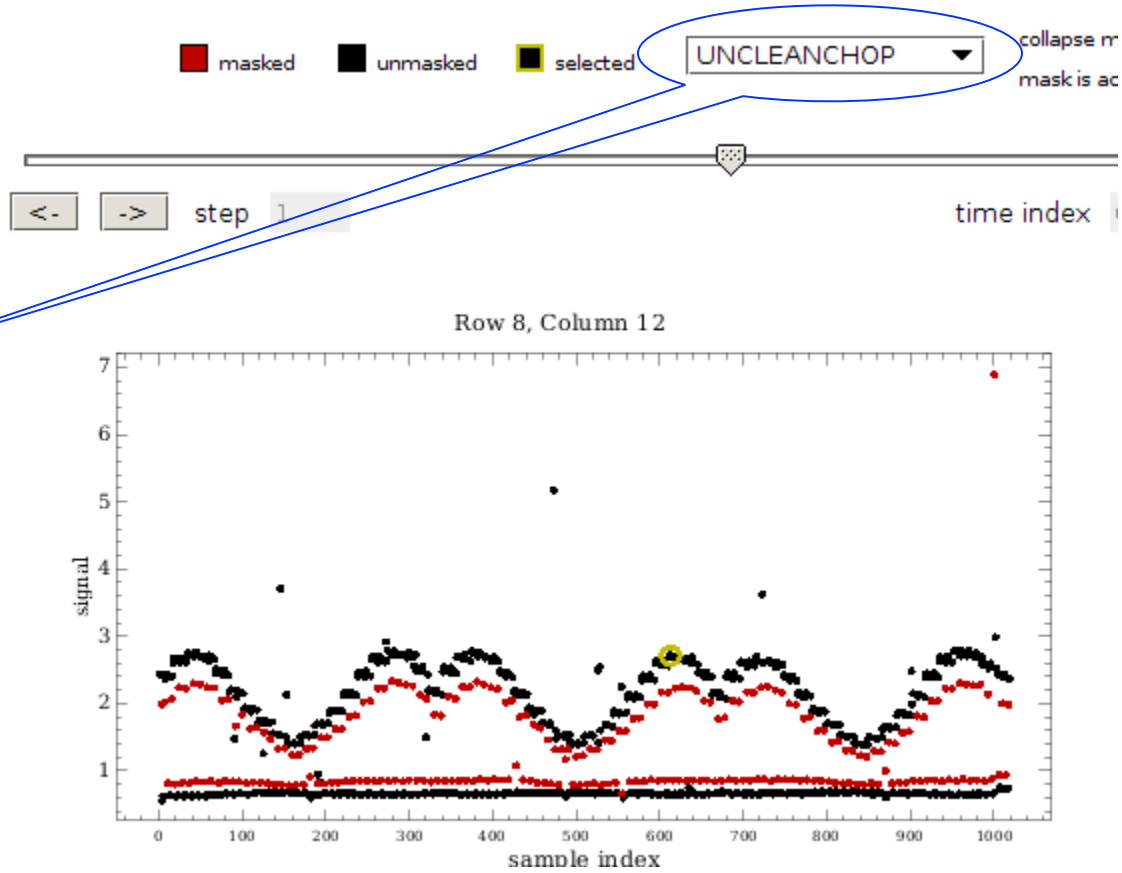
ON signal

OFF signal



# More masks

It is possible to explore other masks




Select unclean chop

In this case, it is clear why there is a second group of points for the ON and OFF positions. These corresponds to signals obtained when the chopper was not yet in the correct position.

Convert signal levels to standard capacitance—Except for very strong signals this would normally be the standard capacitance used

```
392 # Inspect signal vs wavelength
393 #       wavelengths = frame.wave[detector, module,:]
394 #       pwave = PlotXY(wavelengths, flux, line=Style.NONE,titleText="Slice "+str(slice),subtitleText
395 #       del (frame, wavelengths, flux)
396
397
398 # Activate all masks, for all slices
399 slicedFrames = activateMasks(slicedFrames, slicedFrames.get(0).getMaskTypes())
400
401 # Convert the signal level to the reference integration capacitance (which is the smallest one)
402 # used cal file: capacitanceRatios
403 slicedFrames = convertSignal2StandardCap(slicedFrames, calTree=calTree)
404
405 # Derive detectors' response from calibration block
406 # used cal files: observedResponse, calSourceFlux
407 calBlock = selectSlices(slicedFrames,scical="cal").get(0)
408 csResponseAndDark = specDiffCs(calBlock, calTree = calTree)
409
```

Use the calibration block to calculate an observation-specific responsivity and dark current map for your observations. This will be applied later. Note this is a change from HIPE 7 and earlier where a ground-based value was used. The new corrections lead to better absolute calibration.

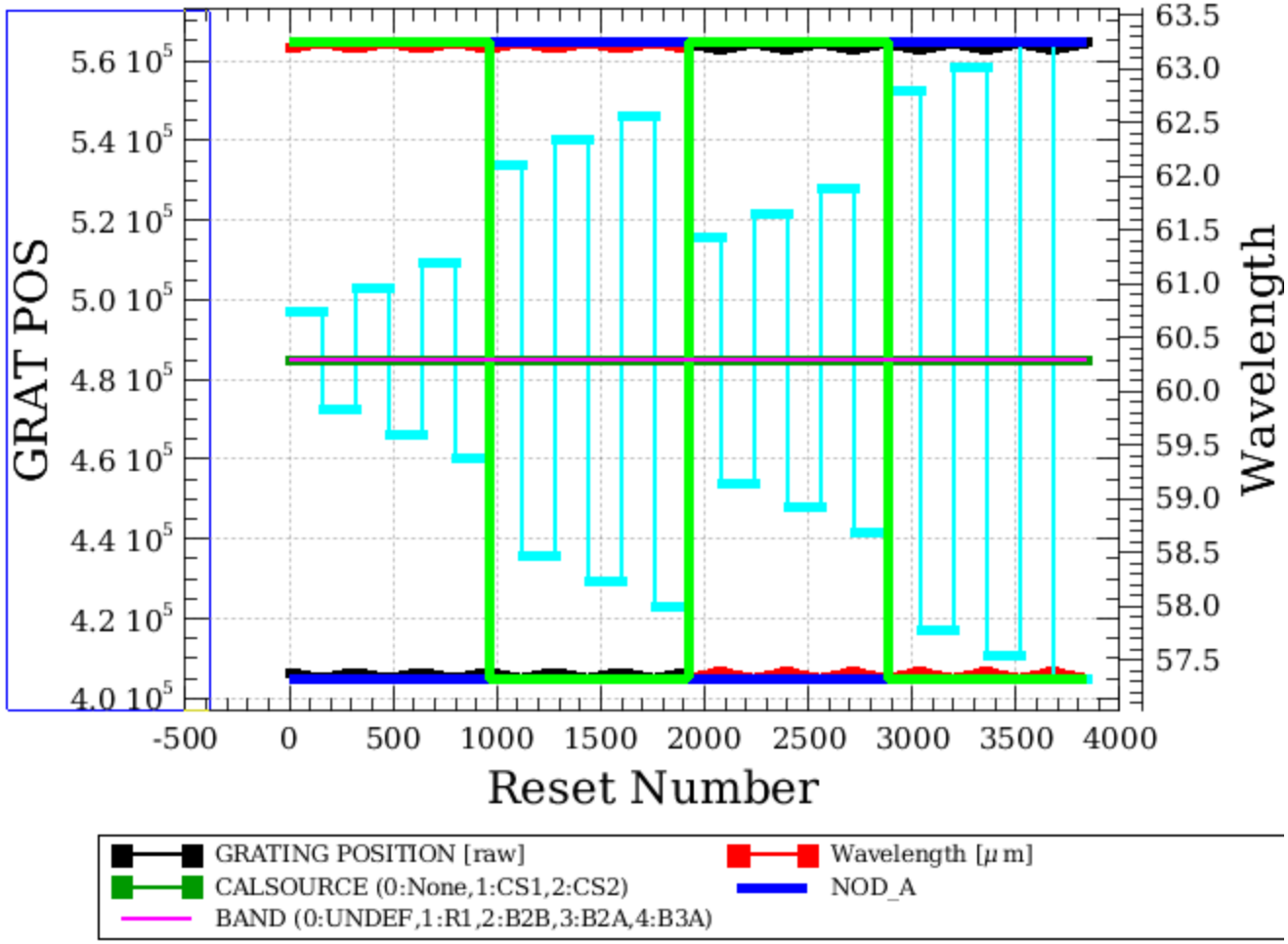
# Chop differentiation

ChopNodEx...RSION.py

```
348
349 # Compute the differential signal of each on-off pair of datapoints, for each chopper cycle
350 # The calibration block is cut out of the slicedFrames, so only the scientific slices remain.
351 slicedFrames = specDiffChop(slicedFrames, scical = "sci", keepall = False, normalize=False)
352
353 if verbose:
354         # Data Structure. Only science blocks are left
355         slicedSummary(slicedFrames)
356         p5 = slicedSummaryPlot(slicedFrames,signal=0)
357         # Detector signal: signal is now differential, with the offs having been
358         # subtracted from the ons (compare to p4 above)
359         p6 = plotSignalBasic(slicedFrames, slice=0)
360
361
```

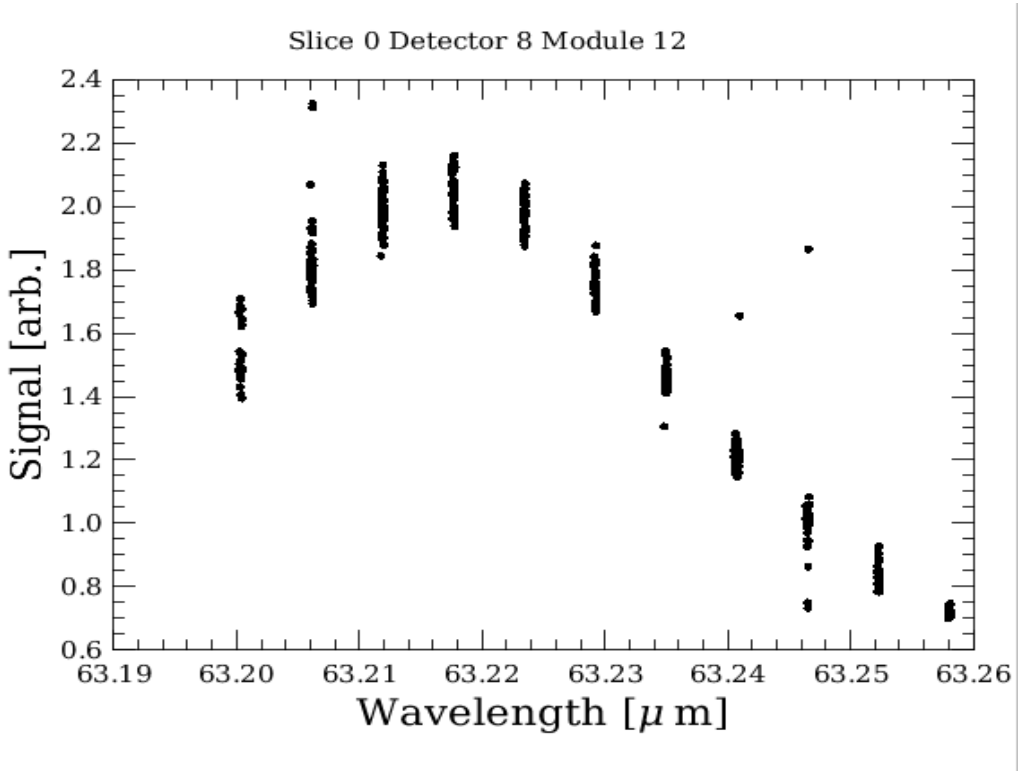
After chop differentiation, the calibration block is excluded from the data

# Chop differentiation



# Chop differentiation

The data are only on the ON position (OFF being subtracted)



**Apply RSRF,**

**Apply response (from cal block)**

```
451
452  # Divide by the relative spectral response function
453  # Used cal files: rsrfR1, rsrfB2A, rsrfB2B or rsrfB3A
454  slicedFrames = rsrfCal(slicedFrames, calTree=calTree)
455
456  # Divide by the response
457  # Use intermediate product from specDiffCs : csResponseAndDark
458  slicedFrames = specRespCal(slicedFrames, csResponseAndDark = csResponseAndDark)
459
460  # Convert the Frames to PacsCubes
461  slicedCubes = specFrames2PacsCube(slicedFrames)
462
463  if verbose: slicedSummary(slicedCubes)
```

Convert data from frames to cube

**Apply Flat field:**

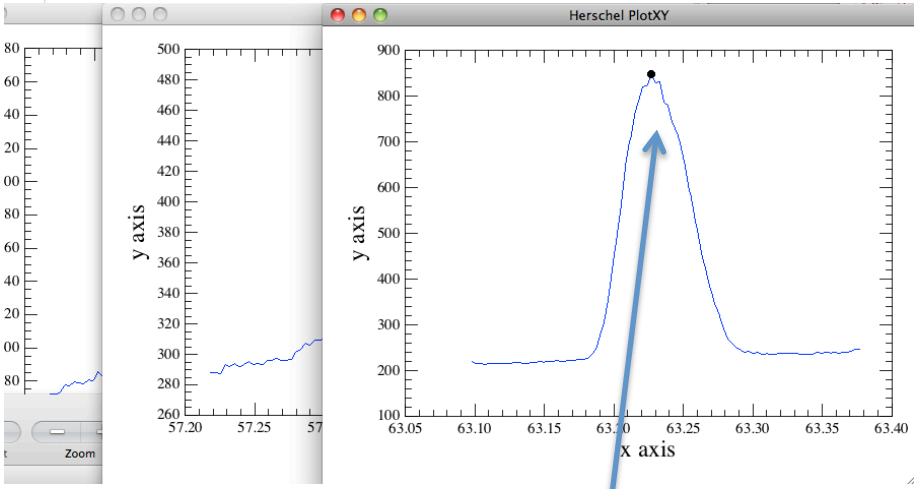
This is a more involved process involving several tasks from the Level 1-2 pipeline which are used here for quickly identify where a bright line is, and remove it for the purpose of the flat field processing. We will not describe in detail all the tasks here since you will learn more about then in the next tutorial PACS 302. If you have a bright line in your dataset, you can pretty much just run this section with the defaults provided. If you suspect that your line is faint, we will explain how to turn off the automatic line identification process. If you are unsure, just run the default pipeline and look carefully at the plots. A more extensive discussion of the full flat fielding process is found in Sect. 3.3 of the PACS data reduction guide found from the "welcome" screen of HIPE under documentation.

First we build a "preliminary" wavegrid, flag outliers, and do a quick rebinning of the data. The aim of this is to allow any bright lines in each spectral slice to be identified for spectral masking. The flat fielding process needs to identify and mask any bright lines in the spectrum before performing the flat-field calculations. If there are no bright lines, there is an option to avoid this process. Note also that if your spectrum contains absorption lines this step must also be avoided. Note that this "preliminary rebinning" will be superseded later by the actually rebinning. However its your "first contact" with your binned spectra and is usually an exciting moment!

```
# 1. Flag outliers and rebin
waveGrid=wavelengthGrid(slicedCubes, oversample=2, upsample=3, calTree=calTree)
slicedCubes = activateMasks(slicedCubes, String1d(["GLITCH","UNCLEANCHOP","NOISYPIXELS","RAWSATURATION","SATU
slicedCubes = specFlagOutliers(slicedCubes, waveGrid, nSigma=5, nIter=1)
slicedCubes = activateMasks(slicedCubes, String1d(["GLITCH","UNCLEANCHOP","NOISYPIXELS","RAWSATURATION","SATU
slicedRebinnedCubes = specWaveRebin(slicedCubes, waveGrid)

# 2. Mask the spectral lines
# First the spectral lines are automatically detected, in every slice.
# widthDetect, expressed in FWHM, and threshold, expressed in local RMS of the
# continuum, are used to detect the spectral lines, in contrast with the local continuum
# widthMask, expressed in FWHM, is used to mask the relevant wavelength ranges
# For more information, print maskLines.__doc__ or consult the help.
# It is recommended to work with verbose=True and carefully inspect the plots
# for what is accepted as a spectral line.
# If necessary, tune the parameters and relaunch slicedMaskLines
# Via the parameter 'lineList', it is possible to skip the automatic detection
# of lines, and instead force the list of lines to be masked.
# This is necessary e.g. for absorption lines.
widthDetect =  2.5
threshold   = 10.
slicedCubesMask = slicedMaskLines(slicedCubes,slicedRebinnedCubes, lineList=[], widthDetect=widthDetect,\
```

```
# 2. Mask the spectral lines
# First the spectral lines are automatically detected, in every slice.
# widthDetect, expressed in FWHM, and threshold, expressed in local RMS of the
# continuum, are used to detect the spectral lines, in contrast with the local continuum
# widthMask, expressed in FWHM, is used to mask the relevant wavelength ranges
# For more information, print maskLines.__doc__ or consult the help.
# It is recommended to work with verbose=True and carefully inspect the plots
# for what is accepted as a spectral line.
# If necessary, tune the parameters and relaunch slicedMaskLines
# Via the parameter 'lineList', it is possible to skip the automatic detection
# of lines, and instead force the list of lines to be masked.
# This is necessary e.g. for absorption lines.
widthDetect =  2.5
threshold   = 10.
slicedCubesMask = slicedMaskLines(slicedCubes,slicedRebinnedCubes, lineList=[], widthDetect=widthDetect,\
widthMask=2.5, threshold=threshold, copy=1, verbose=verbose, maskType="INLINE")
```




Dot means the algorithm found a line peak

For each slice you will get a plot of the spectrum and the black dot indicates that the software has detected a line at that point. It will then mask out an area of the spectrum around that point of widthdetect*FWHM, and use whats left to form the flat field. If the black dot has not found a peak or its in the wrong place then you should consider treating the spectrum as a non-detection. Also absorption lines will not be treated correctly.

Lets assume that all is well and a line is bright and identified correctly in each slice (we will explain the alternative in a moment)

Next look again at the spectra produced. Is there a significant spectral slope to the continuum?
If so, then accept the default for "slopeInContinuum" = 1 (True)
If not, you might consider setting to 0 (false) but it will still work with 1.

Then apply the flatfield correction: (The rest is just clean-up afterwards)

```
# 3. Actual spectral flatfielding
# slopeInContinuum is a boolean. Set it to true for lines existing on a continuum
# with a significant spectral slope.
# For more information, print specFlatFieldLine.__doc__ or consult the help.
slopeInContinuum = 1

slicedCubes = slicedSpecFlatFieldLine(slicedCubesMask, scaling=1, copy=1, maxrange=[50.,230.], slopeInContir

# 4. Rename mask OUTLIERS to OUTLIERS_1 (specFlagOutliers would refuse to overwrite OUTLIERS) & deactivate r
slicedCubes.renameMask("OUTLIERS", "OUTLIERS_1")
slicedCubes = deactivateMasks(slicedCubes, String1d(["INLINE", "OUTLIERS_1"]))
if verbose: maskSummary(slicedCubes, slice=0)

# 5. Remove intermediate results
del waveGrid, slicedRebinnedCubes, slicedCubesMask
# --- End of Spectral Flat Fielding

if verbose: maskSummary(slicedCubes, slice=0)
```

What if you have:

1) No obvious line, or a very weak one not well identified?
2) A known absorption line?

In this case you circumvent the automatic line identification by simply specifying the central wavelength of the expected line in the lineList parameter to the call: (See Section 3.3 of the Pacs Data Reduction Guide accessed through the help in HIPE).

```
# via the parameter 'lineList', it is possible to skip the automatic detection
# of lines, and instead force the list of lines to be masked.
# This is necessary e.g. for absorption lines.
widthDetect =  2.5
threshold   = 10.
slicedCubesMask = slicedMaskLines(slicedCubes,slicedRebinnedCubes, lineList=[63.24], widt
widthMask=2.5, threshold=threshold, copy=1, verbose=verbose, maskType="INLINE")
```

By adding a line (or list of lines e. g. 63.24, 70.23 as parameter, the pipeline will automatically ignore the automatic line ID and will mask around the line position you have given. It will then proceed as normal through the rest of the pipeline.

Congratulations: You have reached Level 1

You may now proceed to web tutorial
PACS-302